46 | AArch64体系：ARM最新编程架构模型剖析

2021-08-23 LMOS

《操作系统实战45讲》 课程介绍 >

讲述：陈晨

时长 21:30 大小 19.70M

你好，我是 LMOS。

在今天，Andriod+ARM 已经成了移动领域的霸主，这与当年的 Windows+Intel 何其相似。之前我们已经在 Intel 的 x86 CPU 上实现了 Cosmos，今天我会给你讲讲 ARM 的 AArch64 体系结构，带你扩展一下视野。

首先，我们来看看什么是 AArch64 体系，然后分析一下 AArch64 体系有什么特点，最后了解一下 AArch64 体系下运行程序的基础，包括 AArch64 体系下的寄存器、运行模式、异常与中断处理，以及 AArch64 体系的地址空间与内存模型。

话不多说，下面我们进入正题。

什么是 AArch64 体系

ARM 架构在不断发展，现在它在各个领域都得到了非常广泛地应用。

自从 Acorn 公司于 1983 年开始发布第一个版本，到目前为止，有九个主要版本，版本号由 1 到 9 表示。2011 年，Acorn 公司发布了 ARMv8 版本。

ARMv8 是首款支持 64 位指令集的 ARM 处理器架构，它兼容了 ARMv7 与之前处理器的技术基础，同样它也兼容现有的 A32（ARM 32bit）指令集，还扩充了基于 64bit 的 AArch64 架构。

下面我们一起来看看 ARMv8 一共定义了哪几种架构，一共有三种。

1.**ARMv8-A（Application）架构**，支持基于内存管理的虚拟内存系统体系结构（VMSA），支持 A64、A32 和 T32 指令集，主打高性能，在我们的移动智能设备中广泛应用。

2.**ARMv8-R（Real-time）架构**，支持基于内存保护的受保护内存系统架构（PMSA），支持 A32 和 T32 指令集，一般用于实时计算系统。

3.**ARMv8-M（Microcontroller 架构）**，是一个压缩成本的嵌入式架构，而且需要极低延迟中断处理。它支持 T32 指令集的变体，主打低功耗，一般用于物联网设备。

今天我们要讨论的 AArch64，它只是 ARMv8-A 架构下的一种执行状态，“64”表示内存或者数据都保存在 64 位的寄存器中，并且它的基本指令集可以用 64 位寄存器进行数据运算处理。

AArch64 体系的寄存器

一款处理器要运行程序和处理数据，必须要有一定数量的寄存器。特别是基于 RISC（精简指令集）架构的 ARM 处理器，寄存器数量非常之多，因为大量的指令操作的就是寄存器。

ARMv8-AArch64 体系下的寄存器简单可以分为以下几类。

1. 通用寄存器

2. 特殊寄存器

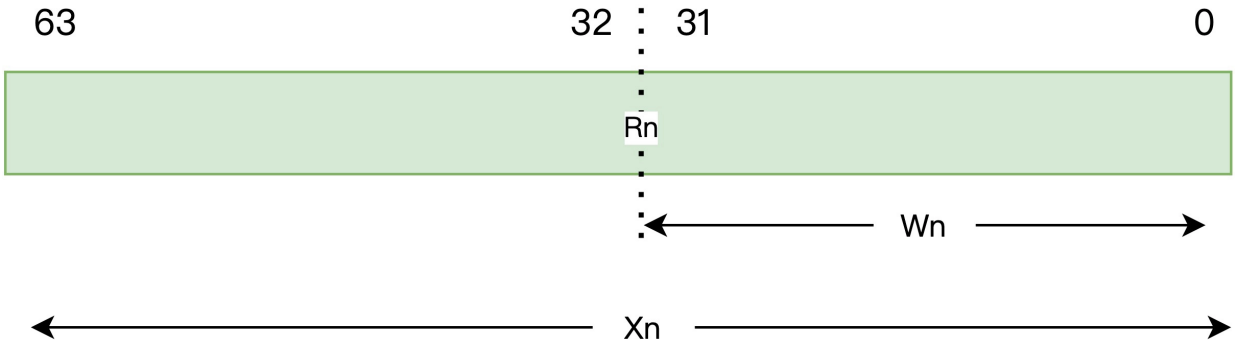
3. 系统寄存器

下面我们分别来看看这三类寄存器。

通用寄存器 R0-R30

首先来看通用寄存器（general-purpose registers），通用寄存器一共为 31 个，从 R0 到 R30，这个 31 个寄存器可以作为全 64 位使用，也可以只使用其中的低 32 位。

全 64 位的寄存器以 x0 到 x30 名称进行引用，用于 32 位或者 64 位的整数运算或者 64 位的寻址；低 32 位寄存器以 W0 到 W30 名称进行引用，只能用于 32 位的整数运算或者 32 位的寻址。为了帮你理解，我还在后面画了示意图。



register_common

通用寄存器中还有 32 个向量寄存器（SIMD），编号从 V0 到 V31。因为向量计算依然是数据运算类的，所以要把它们归纳到通用寄存器中。每个向量寄存器都是 128 位的，但是它们可以单独使用其中的 8 位、16 位、32 位、64 位，它们的访问方式和索引名称如下所示。

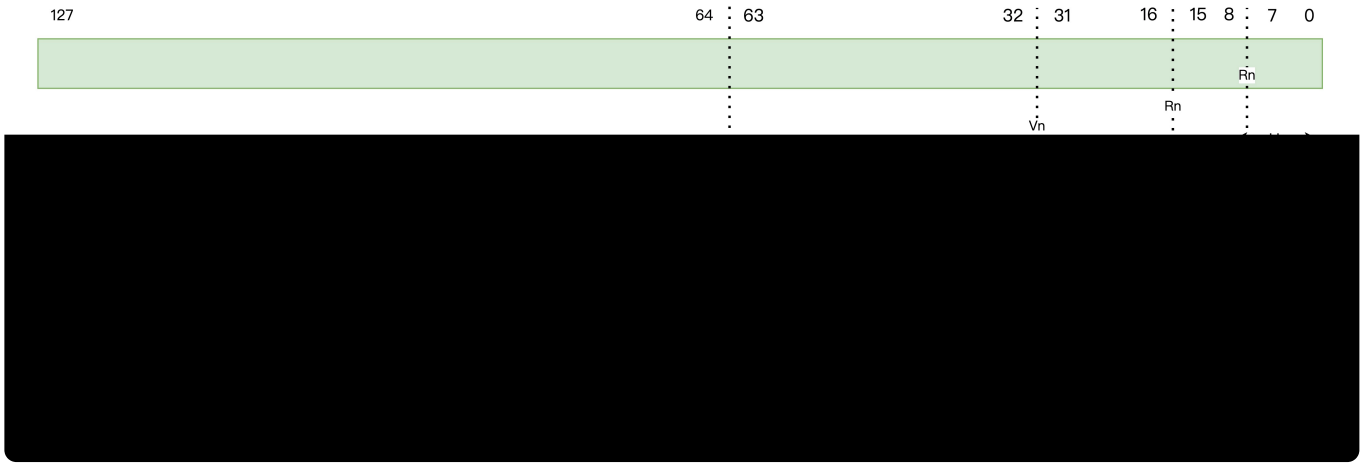
Q0 到 Q31 为一个 128-bit 的向量寄存器 ；

D0 到 D31 为一个 64-bit 的向量寄存器；

S0 到 S31 为一个 32-bit 的向量寄存器；

H0 到 H31 为一个 16-bit 的向量寄存器；

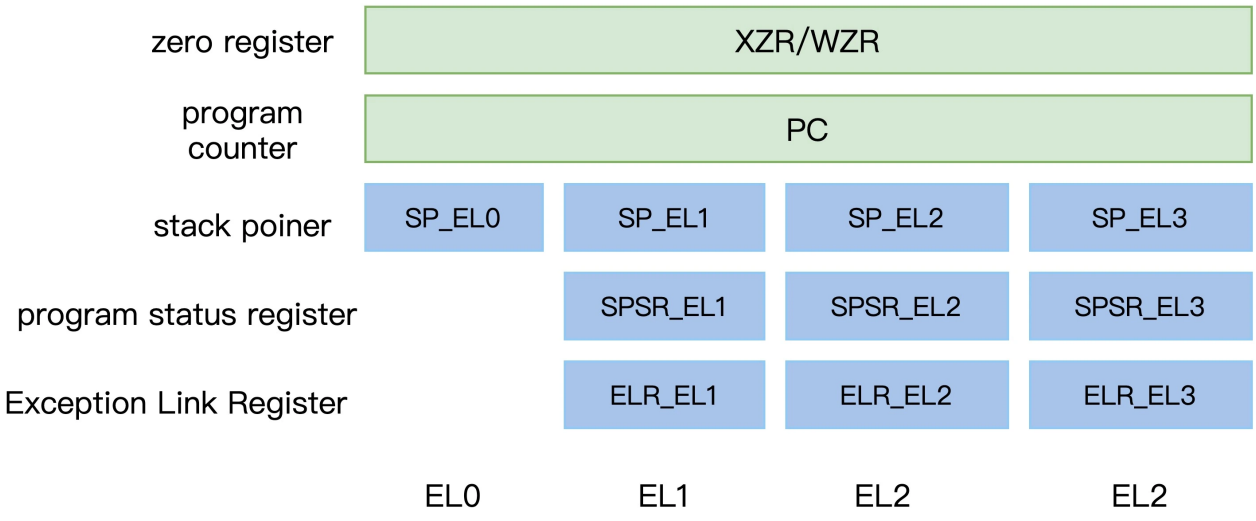
B0 到 B31 为一个 8-bit 的向量寄存器；



register_simd

特殊寄存器

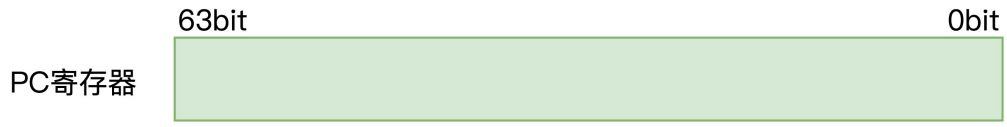
特殊寄存器（spseical registers）比通用寄存器稍微复杂一些，它还可以细分，包括程序计数寄存器（PC），栈指针寄存器（SP），异常链接寄存器（ELR_ELx），程序状态寄存器（PSTATE、SPSR_ELx）等。



special_registers

PC 寄存器

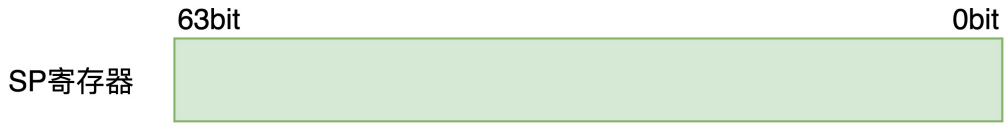
PC 寄存器，保存当前指令地址的 64 位程序计数器，指向即将要执行的下一条指令，CPU 正是在这个寄存器的指引下，一条一条地运行代码指令。在 ARMv7 上，PC 寄存器就是通用寄存器 R15，而在 ARMv8 上，PC 寄存器不再是通用寄存器，不能直接被修改，只可以通过隐式的指令来改变，例如 PC-relative load。



PC寄存器

SP 寄存器

SP 是 64 位的栈指针寄存器，可以通过 **WSP** 寄存器访问低 32 位，在指令中使用 SP 作为操作数，表示使用当前栈指针。C 语言调用函数和分配局部变量都需要用栈，栈是一种后进先出的内存空间，而 SP 寄存器中保存的就是栈顶的内存地址。



SP寄存器

ELR_ELx 异常链接寄存器

每个异常状态下都有一个 ELR_EL 寄存器，ELR_ELx 寄存器是异常综合寄存器或者异常状态寄存器 ，负责保存异常进入 Elx 的地址和发生异常的原因等信息。

该寄存器只有 ELR_EL1、ELR_EL2、ELR_EL3 这几种，没用 ELR_EL0 寄存器，因为异常不会 routing(target) 到 EL0。例如：16bit 指令的异常、32bit 指令的异常、simd 浮点运算的异常、MSR/MRS 的异常。

ELR_ELx寄存器

PSTATE

PSTATE 不是单独的一个寄存器，而是保存当前 PE（Processing Element）状态的一组寄存器统称，其中可访问寄存器有：NZCV、DAIF、CurrentEL（ ）、SPSel。这些属于 ARMv8 新增内容，在 64bit 下可以代替 CPSR（32 位系统下的 PE 信息）。

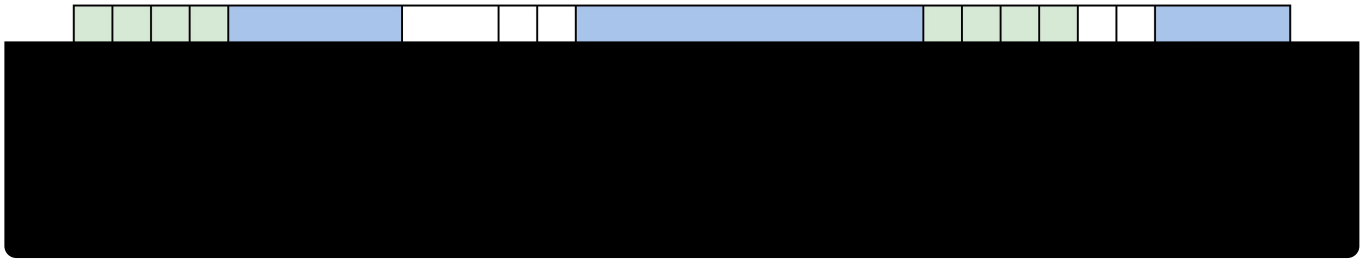
复制代码

```
type ProcState is (
// PSTATE.{N, Z, C, V}： 条件标志位，这些位的含义跟之前AArch32位一样，分别表示补码标志，
bits (1) N, // Negative condition flag
bits (1) Z, // Zero condition flag
bits (1) C, // Carry condition flag
bits (1) V, // oVerflow condition flag
// D表示debug异常产生，比如软件断点指令/断点/观察点/向量捕获/软件单步 等；
// A, I, F表示异步异常标志，异步异常会有两种类型：一种是物理中断产生的，包括SError（系统错
// 另一种是虚拟中断产生的，这种中断发生在运行在EL2管理者enable的情况下：vSError，vIRQ，vF
bits (1) D, // Debug mask bit [AArch64 only]
bits (1) A, // Asynchronous abort mask bit
bits (1) I, // IRQ mask bit
bits (1) F, // FIQ mask bit
// 异常发生的时候，通过设置MDSCR_EL1.SS 为 1启动单步调试机制；
bits (1) SS, // Software step bit
// 异常执行状态标志，非法异常产生的时候，会设置这个标志位，
bits (1) IL, // Illegal execution state bit
bits (2) EL, // Exception Level (see above)
// 表示当前ELx 所运行的状态，分为AArch64和AArch32:
bits (1) nRW, // not Register Width: 0=64, 1=32
// 某个ELx 下的堆栈指针，EL0下就表示sp_el0；
bits (1) SP, // Stack pointer select: 0=SP0, 1=SPx [AArch64 only]
)
```

SPSR_ELx 程序状态寄存器

程序在运行中，处理大量数据，无非是进行各种数学运算，而数学运算的结果往往有各种状态，如进位、结果为 0、结果是负数等，还有程序的运行状态，是否允许中断，CPU 的工作模式，这些信息都保存在程序状态寄存器中，即 PSTATE 中。

但是当 CPU 处理异常时，进程相应的 ELx 状态不同，就要把 PSTATE 状态信息保存在 ELx 状态下对应的 SPSR_ELx 寄存器中。SPSR_ELx 寄存器的格式如下所示。



register_SPSR_ELx

系统寄存器

最后，ARM 的 CPU 上还有一些系统寄存器，用于访问系统配置。

在 EL0 状态下，大多数系统寄存器是不可访问的，但是部分系统寄存器可以在 EL0 状态下进行访问，比如 Cache ID 寄存器（用于 EL0 状态下缓存管理）、调试寄存器（用于代码调试，如 MDCCSR_EL0、DBGDTR_EL0 等）、性能监控寄存器和时钟寄存器等。

ARM-A Arch64 体系下 CPU 的工作模式

其实，AArch64、AArch32 体系都是简称，从严格意义上说，它们应该是处理器的两种执行方式或者状态。AArch64 体系执行 A64 指令集，这个指令集是全 64 位的；AArch32 体系则可以执行 A32 指令集和 T32 指令集（这节课我们不关注这个体系，所以这些指令集暂不深究）。

不管是 AArch64 体系还是 AArch32 体系，ARM CPU 的工作模式并没有差异。为了让你把握重点，我们后面只是以 AArch64 体系为例，探讨 ARM 处理器的工作模式。

工作模式分类

前面我们介绍了 x86 CPU 的工作模式，但是 x86 CPU 的工作模式和 ARM 的 CPU 的工作差别很大，x86 CPU 的工作模式，包括特权级、处理器位宽、内存的访问与保护。

ARM CPU 工作模式则有些不同，究竟有哪些不同呢？

ARM 的 CPU 一共有 7 种不同工作模式，根据权限和状态，以及进入工作模式的方法等方面的不同，我为你用表格的方式做了梳理。

工作模式	说明	是否是特权模式	是否是异常模式
用户 – user 模式	用户程序运行模式	否	否
系统 – system 模式	运行特权级的操作系统任务	是	否
一般中断 – IRQ 模式	普通中断模式	是	是
快速中断 – FIQ 模式	快速中断模式	是	是
管理 – supervisor 模式	提供操作系统使用的一种保护模式，swi 命令状态	是	是
中止 – abort 模式	虚拟内存管理和内存数据访问保护	是	是
未定义指令终止 – undefined 模式	支持通过软件仿真硬件的协处理	是	是

极客时间

工作模式梳理

虽然看起来比较多，但是还是比较好归纳的，在 7 种模式中，除了用户模式之外的模式，被统称为 **Privileged Modes**(特权模式)。

首先，我们大多数的应用程序是运行在用户模式下的，在用户模式下，是不能够访问受保护的系统资源的。此外，应用程序也无法进行处理器模式的切换的。这样就做到了应用程序和内核程序的权力分隔，确保应用程序不能破坏操作系统。

一旦代码的执行流，切换到特权模式下，其代码就可以访问全部的系统资源了，代码也可以随时进行处理器模式的切换。而且只有在特权模式下，CPU 的部分内部寄存器才可以被读写。这里的代码就是指内核代码。

其次，系统模式也是特权模式，代码也是可以访问全部系统资源，也可以随时进行处理器模式的切换，主要供操作系统任务使用。系统模式和用户模式可以访问到的寄存器是同一套的，区别就是它是特权模式，不受用户模式的限制，一般系统模式用于调用操作系统的系统任务。

最后，特权模式下，除系统模式之外的其他五种模式就是异常模式。异常模式一般是在用户的应用程序发生中断异常时，随着特定的异常而进入的，比如之前我们讲过的硬件中断和软件中断，每种异常模式都有对应的一组寄存器，用来保证用户模式下的状态不被异常破坏。这样可以大大减小处理异常的时间，因为不用保存大量用户态寄存器。

处理器如何切换工作模式

前面我们已经了解了 ARM 架构下 CPU 的几种工作模式，那么 CPU 的工作模式是如何切换的呢？

工作模式切换大概分两种情况，一是软件控制，通过修改相应的寄存器或者执行相应的指令；二是当外部中断或是异常发生时，也会导致 CPU 工作模式的切换。

那么当 CPU 发生中断或者异常时，CPU 进入相应的异常模式时，以下工作由 CPU 自动完成。

1. 在异常模式的 R14 中，保存前一个工作模式里，下一条即将执行的指令地址；

2. 将 CPSR 的值复制到异常模式的 SPSR 中；

3. 将 CPSR 的工作模式设为该异常模式对应的工作模式；

4. 令 PC 值等于这个异常模式在异常向量表中的地址，即跳转去执行异常向量表中的相应指令。

处理完中断或者异常，就需要从中断或者异常中返回到发生中断或者异常的位置，继续执行程序。这个从异常工作模式退回到之前的工作模式时，需要由软件来完成后面这两项工作。

1. 将异常模式的 R14 减去一个适当的值（4 或 8）后，赋给 PC 寄存器；

2. 将异常模式 SPSR 的值赋给 CPSR；

好了，以上就是 CPU 切换工作的细节，有了这个基础，接下来我们一起看看 AArch64 体系下 CPU 是如何处理中断或者异常的。

AArch64 体系如何处理中断

现在我们来看看 AArch64 体系是如何处理中断的，首先我们要搞清楚中断和异常的区别，然后了解它们的处理过程，最后再研究一下中断向量表。

异常和中断

有时候，我们习惯于把异常（Exception）和中断（Interrupt）理解成一回事儿。但是对 ARM 来说，官方文档用了 Exception 这个术语来描述广义上的中断，包括异常（Exception）和中断（Interrupt），Exception 和 Interrupt 的执行机制都是一样的，只是触发方式有区别。

这里的异常，切入的视角是处理器**被动接收**到了异常。异常通常表现为错误，比如 CPU 执行了未知指令，但 CPU 明显不能执行这个指令，所以就会产生错误。再比如说，CPU 访问了不能访问的内存，这也是错误的。你会发现，共同点是异常都是同步的，不修改程序下次同样会发生。

而中断对应的视角是处理器**主动申请**，你可以当作是异步的异常，因外部事件产生。中断分为三种，它们分别是 IRQ、FIQ 和 SError。IRQ、FIQ 通常是连接到外部中断信号，当外部设备发出中断信号时，CPU 就能对此作出响应并处理外部设备需要完成的操作。

中断处理

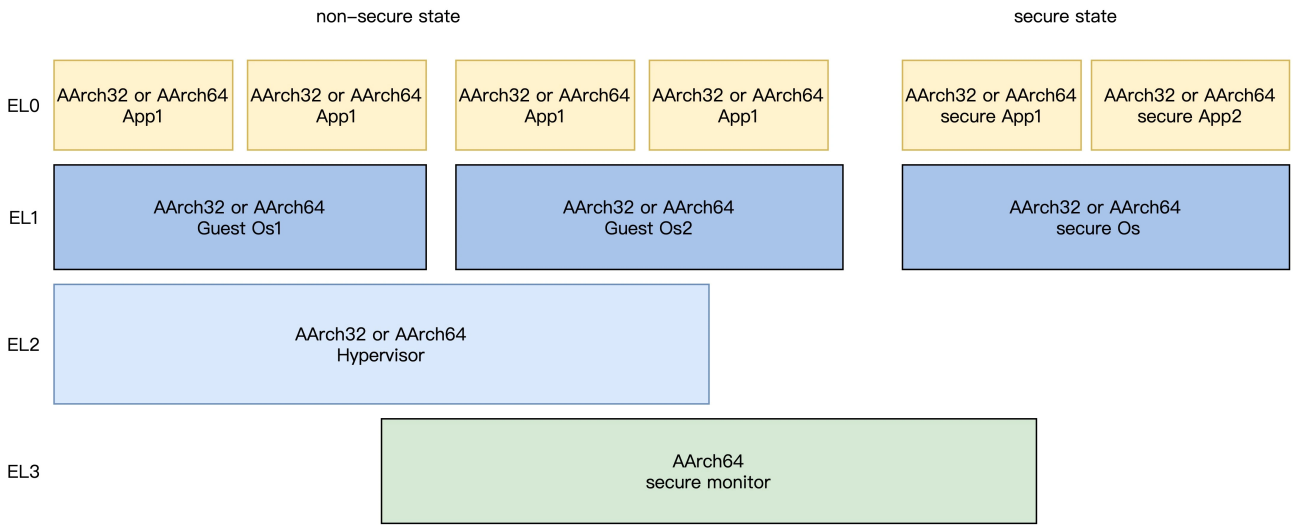
我们在了解中断处理之前，首先要搞明白异常级别。

在全局 ARMV8-A 体系结构中，定义了四个异常级别（Exception Level）从 EL0 到 El3，每个异常级别的权限不同，你不妨想像一下 x86 CPU 的 R3～R0 特权级。

只不过 ARMV8-A 体系结构下 EL0 为最低权限模式，也就是对应用户态，处理的是应用程序；EL1 处理的是 OS 内核层，对应的是内核态；EL2 是 Supervisor 模式，处理的则是可以跑多个虚拟 OS 内核的管理软件，对应的是虚拟机管理态，它是可选的，如 Hypervisor

用于和 virtualization 扩展；EL3 运行的是安全管理（Secure Monitor），处理的是监控态，用于 security 扩展。

开发通用的操作系统内核只需要使用到 EL1，EL2 两个异常级别，我为你画了一幅 EL 模型图，如下所示。



EL模型图

现在我们来看看中断或者异常发生时，EL 级别的切换，这里分为两种情况。

第一种是高级别向低级别切换，这种方式通过修改 PSTATE 寄存器中的值来实现，EL 异常级别就保存在这个寄存器中；第二种是低级别向高级别切换，通过触发中断或者异常的方式进行切换的。

在这两种切换过程中，如果高级的状态是 AArch64，低级的可以是 AArch64 或者 AArch32，也就是可以向下兼容；如果高级的是 AArch32，那么低级的也一定要是 AArch32。

当一个中断或者异常触发后，CPU 的操作流程如下所示。

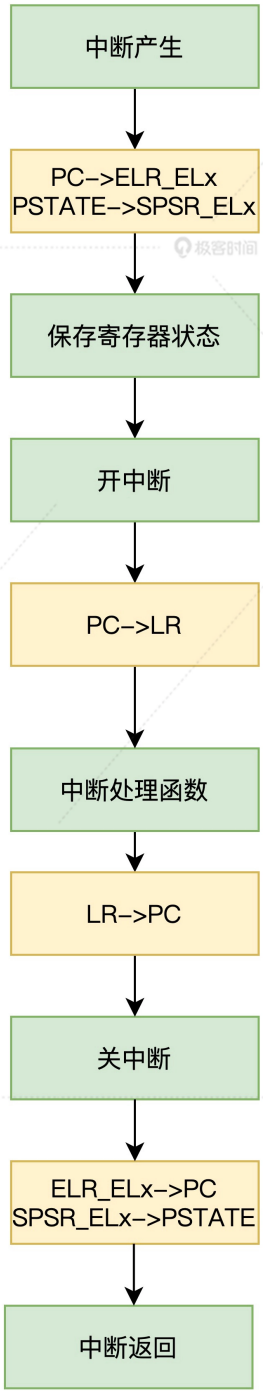
1. 更新 SPSR_ELx 寄存器，即当前的 PSTATE 寄存器的信息存储在 SPSR_ELx 寄存，以便中断结束时恢复到 PSTATE 寄存器。

2. 更新 PSTATE 寄存器以反映新的处理器状态，这个过程中，中断级别可能会发生变化。

3. 发生中断时的下一条指令地址存储在 ELR_ELx 寄存器中，以便中断返回后，能继续运行。

4. 当中断处理完成后，由高级别返回低级别时，需要使用 ERET 指令返回。

下图能帮你更加清楚地理解这一行为。



Interrupt流程

上图已经清楚地展示了，中断或者异常发生时，其中几个关键寄存器是如何保存和恢复的。

中断向量表

当中断或者异常发生后，CPU 进行相应的操作后，必须要跳转到相应的地址开始运行相应的代码，进行中断或者异常的处理，这个地址就是**中断向量**。由于有多个中断或者异常，于是就形成了**中断向量表**。

在 AArch64 中，每个中断或者异常触发时会产生 EL 级别切换。通常在 EL0 级别调用 svc 指令，触发一个同步异常，CPU 则会切换到 EL1 级别；如果在 EL0 级别来了一个 IRQ 或 FIQ，就会触发一个异步中断，CPU 会根据 SCR 寄存器中的中断配置来决定切换 EL1 或 EL2 或 EL3 级别，同时也会区分 EL 级别使用的是 AArch64，还是 AArch32 的指令集。

16 个向量的分类和偏移地址在向量表中的关系如下所示。

Address	Execption type	Description
VBAR_Eln+0x000	Synchronous	Current EL wit SP0
0x080	IRQ/vIRQ	
0x100	FIQ/vFIQ	
0x180	Serror/vSError	
0x200	Synchronous	Current EL wit SPx
0x280	IRQ/vIRQ	
0x300	FIQ/vFIQ	
0x380	Serror/vSError	
0x400	Synchronous	Lower EL using AArch64
0x480	IRQ/vIRQ	
0x500	FIQ/vFIQ	
0x580	Serror/vSError	
0x600	Synchronous	Lower EL using AArch32
0x680	IRQ/vIRQ	
0x700	FIQ/vFIQ	
0x780	Serror/vSError	

上表中分了四个小表，小表中的每一个 entry 由不同的中断的类型（IRQ，FIQ，SError，Synchronous）决定。具体使用哪一个小表由以下几个条件决定。

1. 如果中断发生在同一中断级别，并且使用的栈指针是 SP_EL0，则使用 SP_EL0 这张表。

2. 如果中断发生在同一中断级别，并且使用的栈指针是 SP_EL1/2/3，则使用 SP_EL 这张表。
3. 如果中断发生在较低的中断级别，使用的小表则为下一个较低级别（AArch64 或 AArch32）的执行状态。

有了这些硬件机制的支持，就可以完美支持现代意义中的操作系统了。

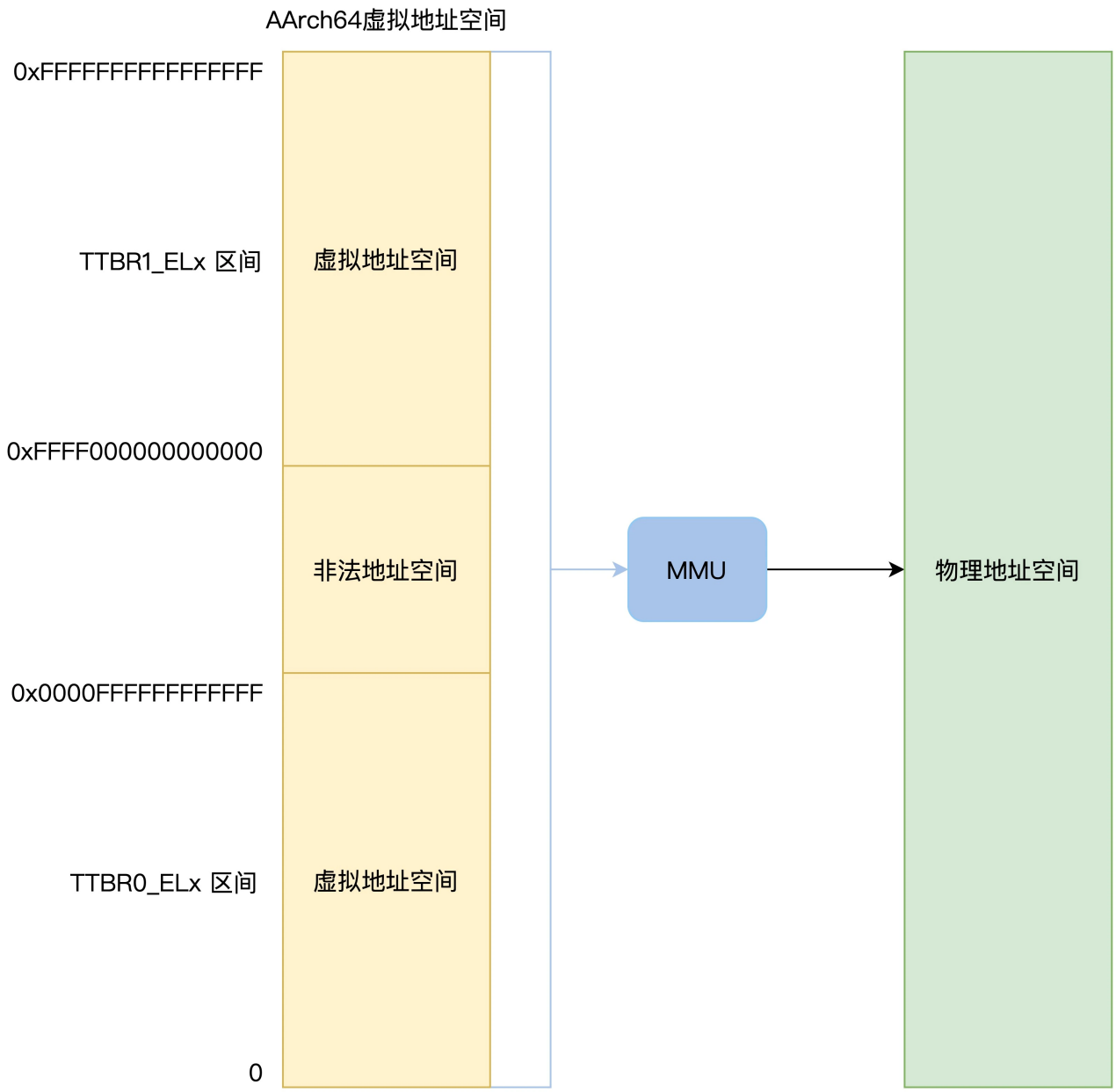
AArch64 体系如何访问内存

无论是操作系统内核代码还是应用程序代码，它们都是放在内存中的，CPU 要执行相应的代码指令，就要访问内存。访问内存有两大关键，**一是寻址，这表现为内存的地址空间；第二个关键点是内存空间的保护，即内存地址的映射和转换**。下面我分别解读一下这两个关键点。

AArch64 体系下的地址空间

对于工作在 AArch64 体系下的 CPU 来说，没有启动 MMU 的情况下，ARM 的 CPU 发出的地址，就是物理地址直接通过这个寻址内存空间。

但是你别以为 AArch64 体系下有 64 位的寄存器，能发出 64 位的地址，就一定能寻址 64 位地址空间的内存。其实实际只能使用 52 位或者 48 位的地址，这里我们只讨论使用 48 位地址的情况。如果启用了 MMU，那么 CPU 会通过虚拟地址寻址，MMU 负责将虚拟地址转换为物理地址，进而访问实际的物理地址空间。这个过程如下图所示。



AArch64虚拟地址空间

上图中可以发现，如果 CPU 发出的虚拟地址在 0x0～0x0000ffffffffffff 范围内，MMU 就会使用 TTBR0_ELx 寄存器指向的地址转换表进行物理地址的转换；如果 CPU 发出的虚拟地址在 0xffff000000000000～0xffffffffffffffff，MMU 使用 TTBR1_ELx 寄存器指向的地址转换表进行物理地址的转换。

究竟虚拟地址是如何转换成物理地址的呢？我们接着往下看。

AArch64 体系下地址映射和转换

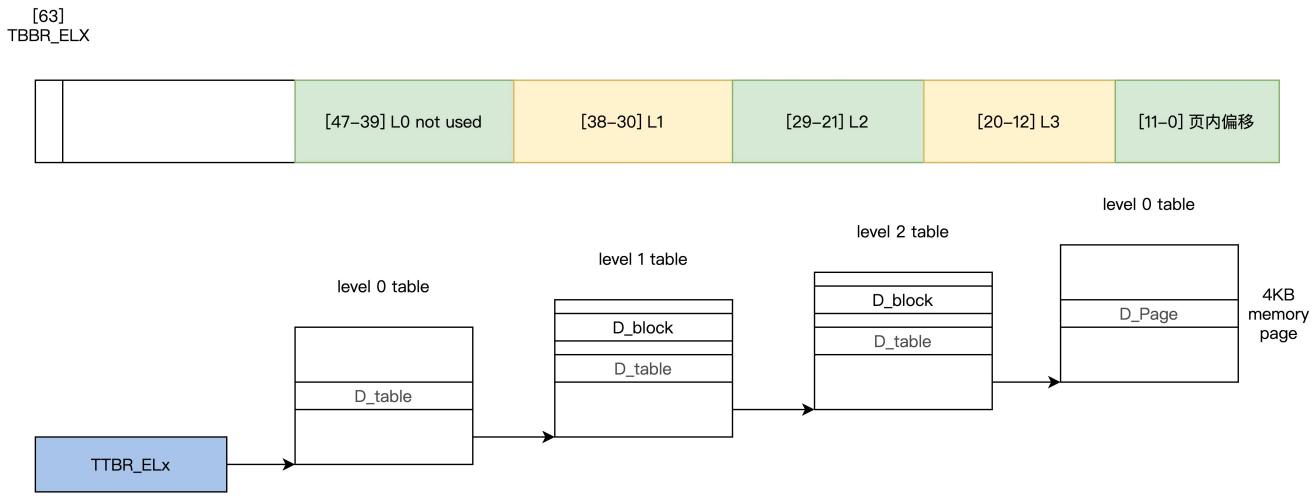
按照我们以往的经验来看，这里肯定是有一张把虚拟地址转化为物理地址的表，给出一个虚拟地址，通过查表就可以查到物理地址。但是实际过程却不是这么简单，在这里通常要

有一个多级的查表过程。

MMU 将虚拟地址映射到物理地址是以页（Page）为单位的，ARMv8 架构的 AArch64 体系可以支持 48 位虚拟地址，并配置成 4 级页表（4K 页），或者 3 级页表（64K 页）。

例如，虚拟地址 0xb7001000~0xb7001fff 是一个页，可能被 MMU 映射到物理地址 0x2000~0x2fff，物理内存中的一个物理页面也称为一个页框（Page Frame）。

那么 MMU 执行地址转换的过程是怎样呢？我们看一看 4K 页表的情况下，虚拟地址转换物理地址的逻辑图。



虚拟地址转化

结合上图我们看到，首先要将 64 位的虚拟内存分成多个位段，这些位段就是用来索引不同级别页表中的 entry 的。那么 MMU 是如何具体操作的呢，一共分五步。

第一步从虚拟地址位段[47:39]开始，用来索引 0 级页表，0 级页表的物理基地址存放在 TTBR_ELx 寄存器中，以虚拟地址位段[47:39]为索引，找到 0 级页表中的某个 entry，该 entry 会返回 1 级页表的基地址。

第二步，接着之前找到的 1 级页表的基地址，现在可以用虚拟地址位段[38:30]索引到 1 级页表的某个 entry，该 entry 在 4KB 页表情况下，返回的是 2 级页表的基地址。

然后到了第三步，有了 2 级页表基地址，就可以用虚拟地址位段[29:21]作为索引找到 2 级页表中的某个 entry，该 entry 返回 3 级页表的基地址。

再然后是第四步，有了 3 级页表基地址，则用虚拟地址位段[20:12]作为索引找到 3 级页表中的某个 entry，该 entry 返回的是物理内存页面的基地址。

最后一步，我们得到物理内存页面基地址，用虚拟地址剩余的位段[11:0]作为索引，就能访问到 4KB 大小的物理内存页面内的某个字节了。

这个过程从 TTBR_ELx 寄存器开始到 0 级页表，接着到 1 级页表，然后到 2 级页表，再然后到 3 级页表，最终到物理页面，CPU 一次寻址，其实是五次访问物理内存。这个过程完全是由硬件处理的，每次寻址时 MMU 就自动完成前面这五步，不需要我们编写指令来控制 MMU，但是我们要保证内核维护正确的页表项。

有了 MMU 硬件转换机制，操作系统只需要控制页表就能控制内存的映射和隔离了。

总结

这节课我们一起了解了 ARM 的 AArch64 体系，它是 ARMV8-A 下的一种执行状态。作为首款支持 64 位的处理器架构，AArch64 体系不只是 32 位 ARM 构架的兼容扩展，还引入了新的 A64 指令集。

处理器想要运行程序、处理数据，离不开各种寄存器。我们学习了 AARch64 下的三类寄存器，包括通用寄存器、特殊寄存器和系统寄存器。

相比 x86 系统，AArch64 的 CPU 工作模式更加多样，一共有七种工作模式。之后，我们分别研究了工作模式切换还有基于 EL0-3 的异常中断处理，以及 AArch64 下的内存架构和访问方式。访问内存，你重点要掌握的是访问内存的两大关键点，**一是寻址，二是内存空间的保护**。

自从 2011 年 ARM 发布首款支持 64 位的 ARMv8 版本后，到现在已经过去了十年。在今年 ARM 也宣布了下一代芯片架构 ARMv9 的部分技术细节，并称□其为十年来最大的创新，也将是未来十年内千亿级别芯片的基础，其在 CPU 性能、安全性、AI 支持上有了显著提升。

但是 ARMv9 不会像 ARMv7 到 ARMv8 的根本性的执行模式和指令集的变化，ARMv9 继续使用 AArch64 作为基准指令集，但是在其功能上增加了一些非常重要的扩展，ARMv9 开发的处理器预计将在 2022 年正式面世，让我们拭目以待！

思考题

请问，ARMv8，有多少特权级？每个特权级有什么作用？

欢迎你在留言区记录你的思考，也欢迎把这节课分享给有需要的朋友。

我是 LMOS，我们下节课见。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 45 | ARM新宠：苹果的M1芯片因何而快？

下一篇 用户故事 | 成为面向“知识库”的工程师

更多课程推荐

精选留言 (4)

写留言

neohope
2021-08-25

非特权状态EL0：运行用户应用程序，无论安全状态还是非安全状态；
特权El1：安全状态下运行操作系统OS；非安全状态下运行虚拟机GuestOS；
特权EL2：安全状态下没有这一级别；非安全状态下运行Hypervisor，为GuestOS提供支持；
特权EL3：运行Secure monitor，管理安全状态与非安全状态的切换；

展开

作者回复: 正确的

2

pedro
2021-08-23

2个特级权，分别是用户态和特权态，但却有7种工作模式，除了用户模式，其它均在特权态，特权态拥有全部权限，可访问一切资源。

信息量太大了，真就南孚聚能环，一节更比六节强！

展开

作者回复: 3级

1

苏流郁宓
2021-08-23

有了 MMU 硬件转换机制，操作系统只需要控制页表就能控制内存的映射和隔离了。
可以理解成ARM处理器对内存利用权限的限制更严格，非特定的特权模式，仅在系统模式下，运行在ARm上的系统不能直接修改内存地址，从而进行内存保护。而是通过mmu页表的变动来影响内存数据（内存地址）的变动吗？

展开

作者回复: 最后一句不准确

苏流郁宓

2021-08-23

3个特权级，6个特权模式？

展开

作者回复: 不对